Memory and Addresses

Memory is just a sequence of byte-sized storage devices.

The bytes are assigned numeric addresses, starting with zero, just like the indexing of the cells of an array.

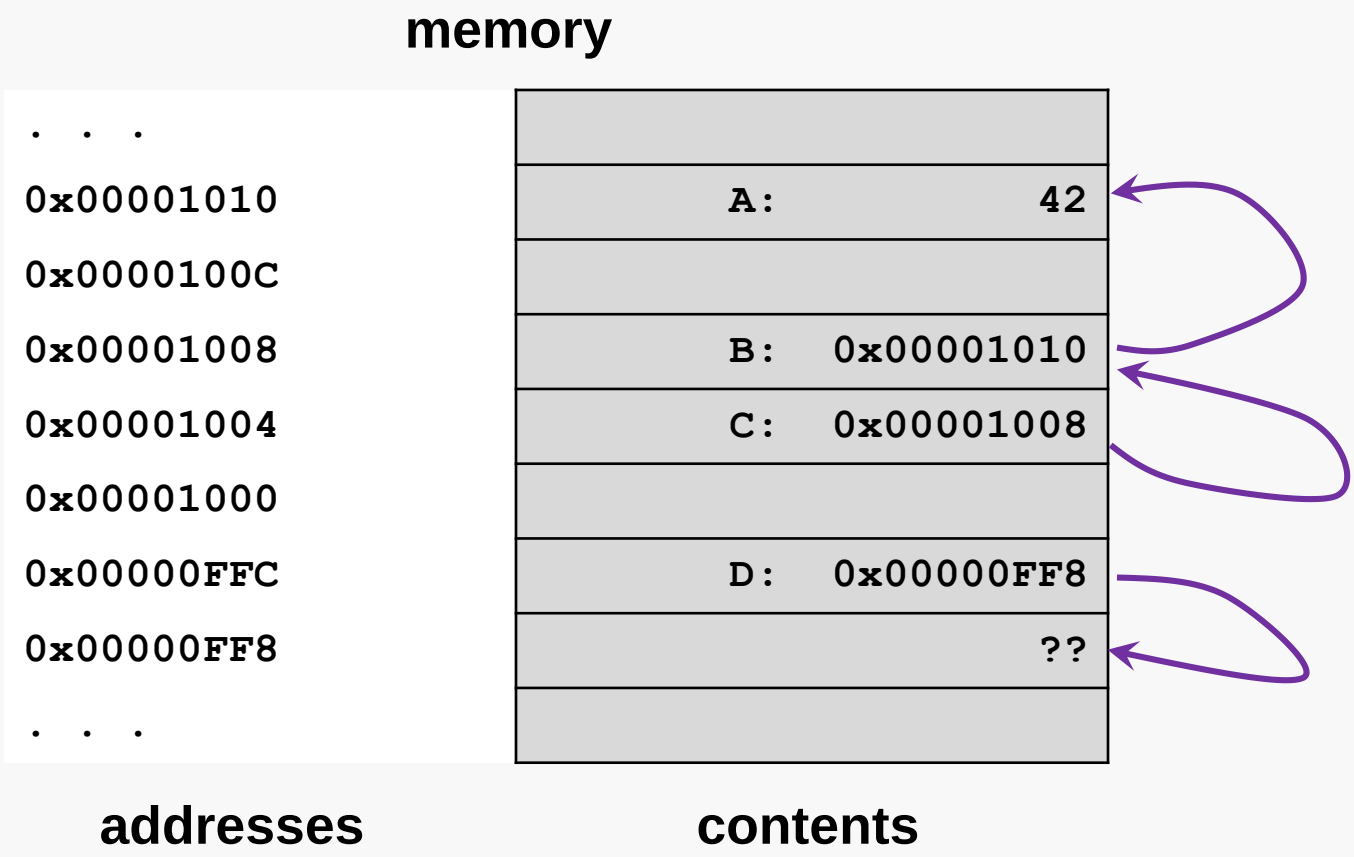
It is the job of the operating system (OS) to:

- manage the allocation of memory to processes
- keep track of what particular addresses each process is allowed to access, and how
- reserve portions of memory exclusively for use by the OS
- enforce protection of the memory space of each process, and of the OS itself
- do all this as efficiently as possible

Pointer Concepts

pointer a variable whose value is a memory address

pointee a value in memory whose address is stored in a pointer; we say the pointee is the *target* of the pointer



Pointer Concepts

Since memory addresses are essentially just integer values, pointers are the same width as integers.

A pointer has a type, which is related to the type of its target.

Pointer types are simple; there is no automatic initialization.

A pointer may or may not have a logically valid target.

Given a pointer that has a valid target, the target may be accessed by *dereferencing* the pointer.

A pointee may be the target of more than one pointer at the same time.

Pointers may be assigned and compared for equality, using the usual operators.

Pointers may also be manipulated by incrementing and decrementing, although doing so is only safe under precisely-defined circumstances.

By convention, pointers without targets should be set to 0 (or `NULL`).

C Syntax: Declaring Pointers

Declarations:

```
int*  p1 = NULL;     // declaration of pointer-to-an-int

char *p2 = 0;        //    pointer-to-a-char

int **p3 = NULL;     //    pointer-to-a-pointer-to-an-int
```

One syntax gotcha:

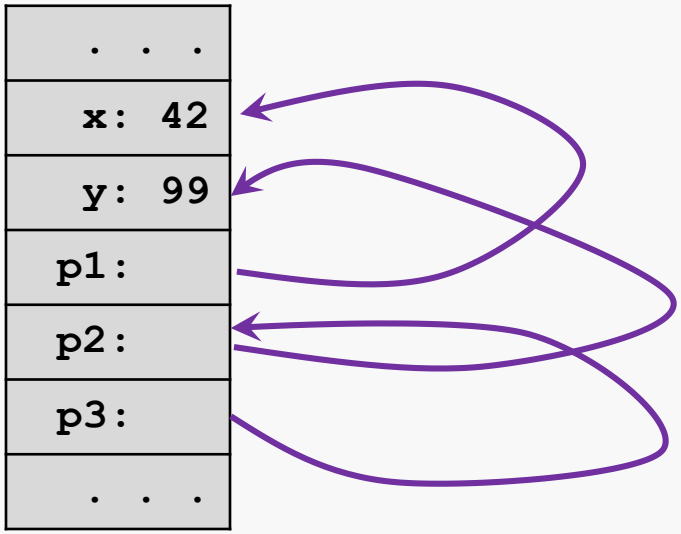
```
int*  q1 = NULL,
      q2 = NULL;     // q2 is an int, not a pointer!
```

```
int  *q1 = NULL,
     *q2 = NULL;     // q1 and q2 are both pointers
```

C Syntax: *address-of* Operator

`&X` returns the address of the object `X`; the *address-of operator*

```
int  x = 42,
     y = 99;

int*  p1 = &x;     // p1 stores address of variable x
int*  p2 = &y;     // p2 stores address of variable y

int** p3 = &p2;    // p3 stores address of variable p2
```



C Syntax: *dereference* Operator

`*P` names the target of the pointer `P`; the *dereference operator*

```
int  x = 42, y = 99;

int*  p1 = &x;     // p1 stores address of variable x
int*  p2 = &y;     // p2 stores address of variable y

int** p3 = &p2;    // p3 stores address of variable p2

int aa = *p1;      // aa stores value of the target of p1, 42
*p1 = 10;          // the target of p1, which is x, stores 10
int bb = *p3;      // illegal: *p3 is a pointer-to-int but bb
                   //    is just an int
int bb = **p3;     // bb stores value of the target of the
                   //    target of p3; p3 points to p2 and
                   //    p2 points to y, so bb gets value 99
```

Understanding C Syntax

```
int  Z = 42;
int *X = &Z;
```

```
X           // refers to X, type is int*
```

```
&X          // refers to address of X, type is int**
```

```
*X          // refers to target of X, type is int
```

```
*&X         // refers to target of address of X, which is just… X
```

```
&*X         // refers to address of target of X, which is just…
            //    the value of X
            // (only makes sense syntactically if X is a pointer)
```

C Example

```
int main() {

   int  x = 42, y = 99;

   int*  p1 = &x;    // p1 stores address of variable x
   int*  p2 = &y;    // p2 stores address of variable y

   int** p3 = &p2;   // p3 stores address of variable p2

   int aa = *p1;     // aa stores value of the target of p1, 42
   *p1 = 10;         // the target of p1, which is x, stores 10

   int bb = **p3;    // bb stores value of the target of the
                     //    target of p3; p3 points to p2 and
                     //    p2 points to y, so bb gets value 99
   return 0;
}
```

C View

```
int main() {
    . . .

    int*  p1 = &x;     // p1 is assigned the address of variable x
    int*  p2 = &y;     // p2 is assigned the address of variable y
    . . .
}
```

`&x`	the address of `x`
`x`	is an `int`
`&x`	is an `int*`
`&y`	the address of `y`

C View

```
int main() {
    . . .

    int** p3 = &p2;    // p3 is assigned the address of variable p2
    . . .
}
```

`&p2`	the address of `p2`
`p2`	is an `int*`
`&p2`	is an `int**`

C View

```
int main() {
    . . .

    int aa = *p1;      // aa is assigned value of the target of p1;
                       // p1 points to x;
                       // x has the value 42
    . . .
}
```

`*p1`	the target of `p1`
`p1`	points to `x`
`x`	has the value `42`
`aa`	is assigned `42`

Value of `*p1`

= value of target of `p1`

= value of `x`

= `42`

C View

```
int main() {
    . . .

    *p1 = 10;       // the target of p1, which is x,
                    //  is assigned the value 10
    . . .
}
```

C View

```
int main() {
    . . .

    int bb = **p3;     // bb stores value of the target of the
                       //    target of p3; p3 points to p1 and
                       //    p1 points to x, so bb gets value 99
    . . .
}
```

`*p3`	the target of `p3`
`p3`	points to `p2`
`p2`	points to `y`
`y`	has the value `99`

Value of `**p3`

= value of target of target of `p3`

= value of target of `p2`

= value of `y`

= `99`

Pointer Comparisons

Pointers may be compared using the usual relational operators.

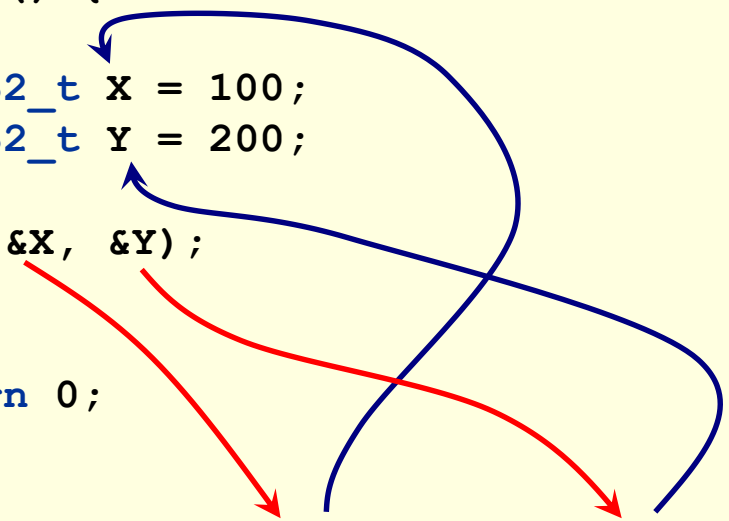
```
p1 == p2          Do p1 and p2 have the
                  same target?
```

```
p1 < p2           Does p1 point to
                  something "below" p2?
```

```
*p1 == *p2        Do the targets of p1 and
                  p2 have the same value?
```

Pointers as Parameters

```
#include <stdint.h>

int main() {

    uint32_t X = 100;
    uint32_t Y = 200;

    Swap(&X, &Y);


    return 0;
}

void Swap(uint32_t* A, uint32_t* B) {

    uint32_t Temp = *A;                 // Temp = 100
    *A = *B;                            // X = 200
    *B = Temp;                          // Y = 100
}
```



The *pass-by-pointer* protocol provides a called function with the ability to modify the value of the caller's variable.

Evil: Dangling Pointers and Aliases

The most common source of errors with direct pointer use is to dereference a pointer that does not have a valid target:

```
int *A;
*A = 42; // A never had a target
```

```
int *A = NULL;
if ( A != NULL )  // used correctly, NULL
   *A = 42;       //   lets us check
```

```
int *A = malloc( sizeof(int) );   // A has a target
int *B = A;                       // B shares it; alias
free(A);                          // neither has a target
*B = 42;                          // ERROR
```

Evil: Dangling Pointers and Aliases

What about doing this:

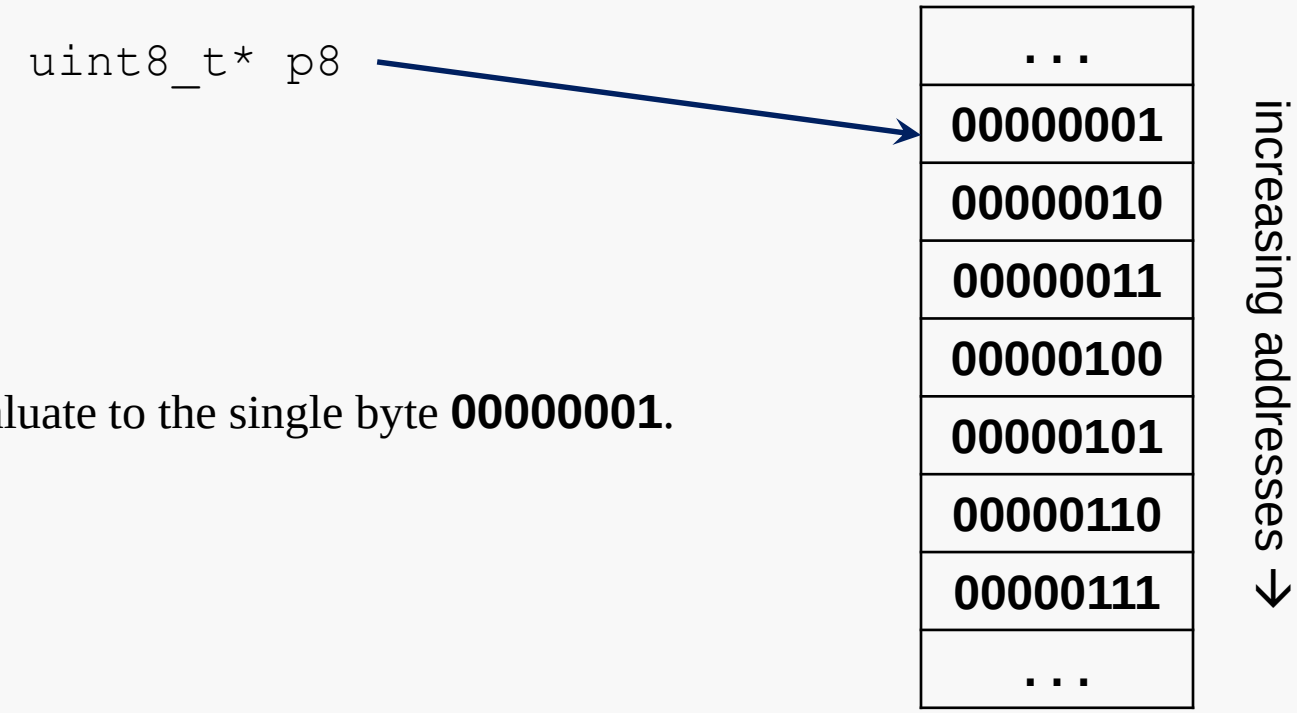
```
int *A;
if ( A != NULL )  // used correctly, NULL
   *A = 42;       //   lets us check
```

Or this:

```
void f(int *A) {
   if ( A != NULL )
      *A = 42;
}
```

Pointers and Raw Memory

Suppose that we have a region of memory initialized as shown below, and a pointer `p` whose target is the first byte of the region:

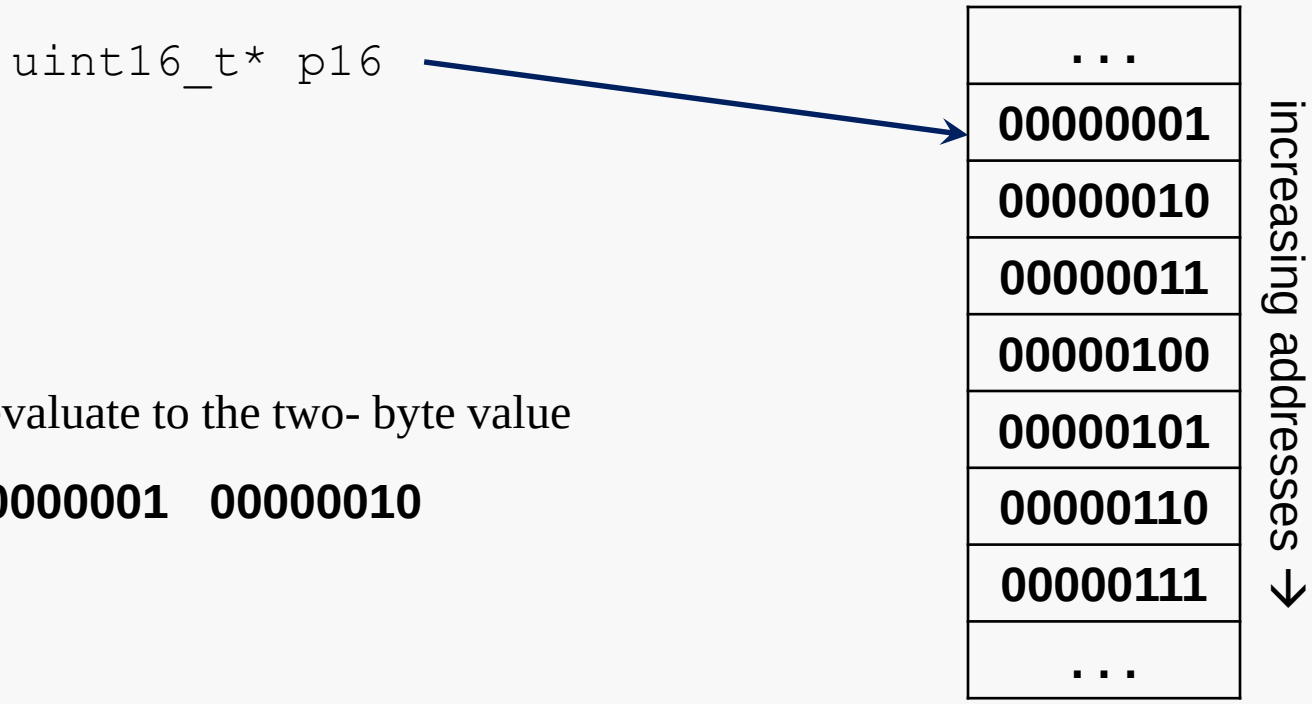


Then `*p8` would evaluate to the single byte **00000001**.

Memory contents are shown in binary

Pointers and Raw Memory

Now suppose that we have a region of memory initialized as shown below, and a pointer `p16` whose target is the first byte of the region:



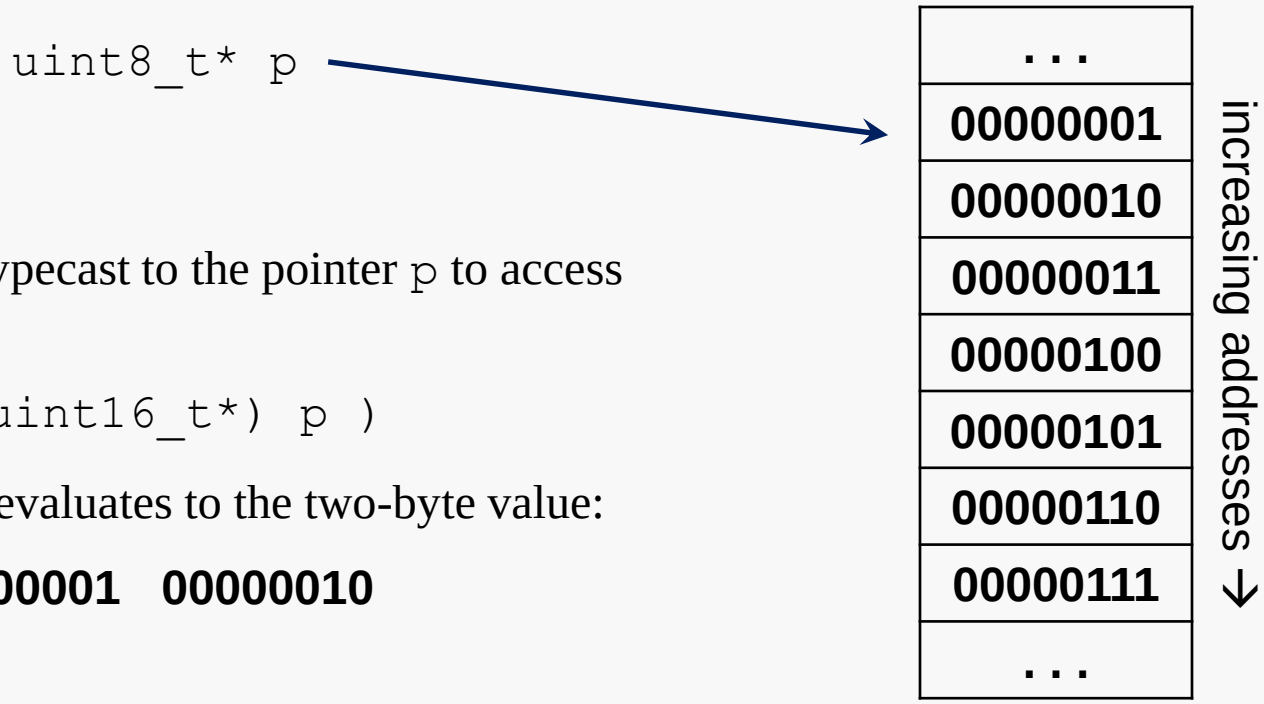
Then `*p16` would evaluate to the two- byte value

000000001 00000010

Memory contents are shown in binary

Pointer Casts

Now suppose that we have a region of memory initialized as shown below, and a pointer `p` whose target is the first byte of the region:



Then we can apply a typecast to the pointer `p` to access two bytes:

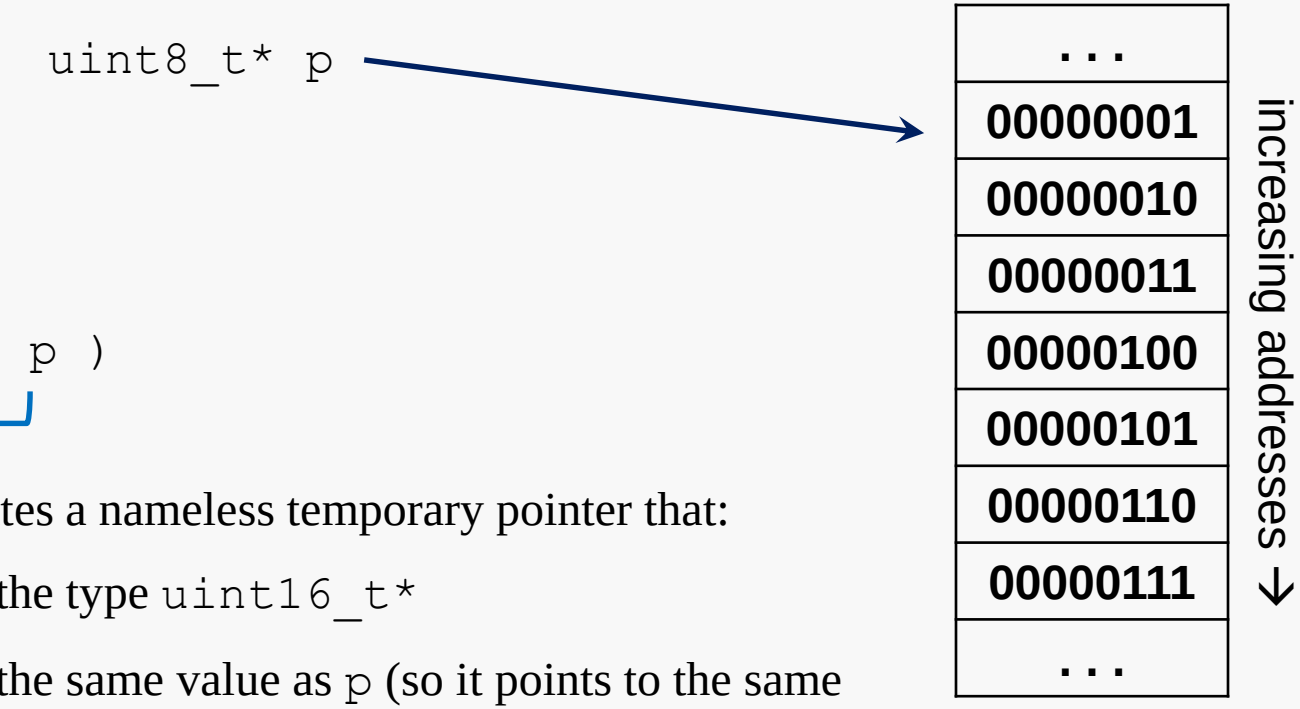
```
*( (uint16_t*) p )
```

The expression above evaluates to the two-byte value:

000000001 00000010

Memory contents are shown in binary

Understanding a Cast




```
*( (uint16_t*) p )
```

This creates a nameless temporary pointer that:

- has the type `uint16_t*`
- has the same value as `p` (so it points to the same target as `p`)

This does not change anything about `p`.

Memory contents are shown in binary

Pointers and Raw Memory Accesses

To generalize, size matters: